1. High-Level Concept

`SWNet4` implements a **compact, Walsh–Hadamard-based neural network** designed to avoid the *spectral bias* of the fast Walsh–Hadamard transform (FWHT).
The core idea is to:

1. **Randomize the input and output Hadamard basis** with *permuted Thue–Morse sign sequences* generated from **Weyl sequences**,

2. Use the FWHT as a **structured dense mixing layer**,

3. Interleave it with **multi-width-4 CReLU blocks** (concatenated ReLU, positive & negative activations) for non-linearity and learnable parameters.

This combination produces a network that mixes globally like dense linear layers, learns locally and uses permuted *Thue–Morse sign sequences* for spectral debiasing where completely random sequences would harm generalization.

---

2. Constructor Details

Parameters

`vecLen` (vector length), `depth` (number of FWHT + CReLU stages), `qIn/qOut` (Thue–Morse step parameters), and learning rate `rate`.

Steps Performed

1. **Gaussian Weight Initialization**

    - Each layer's weight vector `wts[i]` is first filled with uniform random floats in `[-1, 1]`.

    - Applying `WHT.whtN` to this uniform vector yields **approximately Gaussian-distributed weights** (by the central-limit effect of the Hadamard transform).

2. **Permuted Thue–Morse Flip Vectors**

- `flipIn` and `flipOut` are built from **Weyl sequences** that drive a *permuted Thue–Morse* bit-parity generator.
- Each float is encoded with a sign bit from the Thue–Morse pattern, giving a deterministic ±1 sequence.
- These sequences have two key properties:
  - **Highly random w.r.t. Walsh basis rows** → breaks FWHT's tendency to concentrate energy ("spectral bias").
  - **Only mildly random on natural signals** → preserves much of the original structure when applied to images or other real data.

---

3. Forward Pass (`recall`)

The forward signal path can be summarized:

```
Input
  → × flipIn (permuted Thue–Morse signs)
  → FWHT (global dense mixing)
  → [CReLU width-4 layer + FWHT] × depth
  → × flipOut (permuted Thue–Morse signs)
  → Output
```

- **FWHT ("one-to-all")**: Each output is a signed sum of all inputs, acting like a fast structured dense linear transform of complexity O(n log n).
- **CReLU width-4 block (`layer4`)**:
  - Processes 4-element chunks.
  - For each element, selects a positive or negative 4-weight branch depending on sign—equivalent to a *concatenated ReLU* (positive/negative path).
  - Outputs another 4-vector `(a,b,c,d)` per chunk.

- By following each CReLU with another FWHT, the network “knits” many width-4 micro-layers into a single wide, globally connected layer.

---

4. Training (`train`)

1. **Forward evaluation** into `work`.
2. **Error vector construction**:
   `work[i] = flipOut[i] * (target[i] - work[i]) * rate`.
3. **Inverse pipeline**:
   - Apply FWHT to the error (its own inverse).
   - `layer4Back` performs:
     - **Standard SGD weight update**: `w += v * e` for the active sign branch.
     - **Error backpropagation** through the 4-wide CReLU.

Thus training is ordinary stochastic gradient descent, but expressed in a custom Hadamard/CReLU setting.

---

5. Architectural Rationale

Feature	Purpose
Permuted Thue–Morse flips	Counteracts FWHT’s spectral bias while preserving input statistics.
Hadamard transforms	Provide cheap O(n log n) dense mixing, orthogonal and energy preserving.
Gaussian weight init via FWHT	Matches the normal assumptions of gradient-based training.

Width-4 CReLU	Adds learnable, sign-sensitive non-linearity with compact parameter count.

This yields a **fast, hardware-friendly network** with rich representational power despite small parameter counts.

---

6. Computational Characteristics

- **Complexity**: O(depth × vecLen log vecLen) per forward or backward pass.
- **Memory**: `8 * vecLen * depth` floats for weights + buffers.
- **Constraints**: `vecLen` must be a multiple of 4 and typically a power of two for the FWHT.

---

7. Strengths and Considerations

- **Strengths**
  - Breaks spectral bias common in FWHT-based models.
  - Deterministic but “random-looking” sign flips improve generalization.
  - Excellent fit for FPGA/ASIC due to Hadamard structure.
- **Limitations**
  - No bias terms or normalization layers.
  - Training stability depends strongly on `rate`.
  - Design is specialized; benefits appear when input dimension is large and power-of-two.

---

8. Summary

`SWNet4` is a **specialized neural architecture** that integrates:

- **Permuted Thue–Morse/Weyl sign-flip sequences** for controlled randomness,
- **Fast Walsh–Hadamard transforms** for global, dense mixing,
- **Width-4 CReLU micro-layers** for nonlinearity and learnable parameters,
- **Straightforward SGD training** with backprop through each CReLU block.

The result is a compact, mathematically elegant network that leverages structured randomness to overcome FWHT spectral bias while maintaining high computational efficiency and a clear training path.

Code (Java):

```
public final class SWNet4 {
   final int vecLen;
   final int depth;
   final float[][] wts;
   final float[][] sur;
   final float[] work;
   final float[] flipIn;
   final float[] flipOut;
   float rate;
   public SWNet4(int vecLen, int depth,int qIn,int qOut, float rate) {
      this.vecLen = vecLen;
      this.depth=depth;
      this.rate=rate;
      wts = new float[depth][8*vecLen];
      sur=new float[depth][vecLen];
      work=new float[vecLen];
      flipIn=new float[vecLen];
      flipOut=new float[vecLen];
      Rand rn=new Rand();
      for (int i = 0; i < depth; i++) {
         for(int j=0; j<wts[i].length; j++) {
            wts[i][j]=rn.nextFloatSym()*0.8660254f;
         }
         WHT.whtN(wts[i]);
      }
      int rIn=0,rOut=0,one=Float.floatToRawIntBits(1f);
      for(int i=0; i<vecLen; i++) {
         rIn+=qIn;
         rOut+=qOut;
         flipIn[i]=Float.intBitsToFloat(one|(Integer.bitCount(rIn&(vecLen-1))<<31));
         flipOut[i]=Float.intBitsToFloat(one|(Integer.bitCount(rOut&(vecLen-1))<<31));
      }
   }

   public void recall(float[] result, float[] input) {
      VecOps.multiply(result,input,flipIn);
      WHT.whtN(result);
      for (int i=0; i < depth; i++) {
         if(result==work) System.arraycopy(work,0,sur[i],0,vecLen);
```

```
        layer4(result,wts[i]);
        WHT.whtN(result);
    }
    VecOps.multiply(result,result,flipOut);
}


public void train(float[] target, float[] input) {
    recall(work,input);
    for (int i=0; i<vecLen; i++) {
        work[i]=flipOut[i]*(target[i]-work[i])*rate;
    }
    for (int i = depth-1; i >=0; i--) {
        WHT.whtN(work);
        layer4Back(sur[i],wts[i],work);
    }
}

public void layer4(float[] x,float[] w) {
    for(int i=0; i<vecLen; i+=4) {
        int k=8*i;
        float a,b,c,d;
        float v0=x[i],v1=x[i+1],v2=x[i+2],v3=x[i+3];
        if(v0<0f) {
            a=v0*w[k];
            b=v0*w[k+1];
            c=v0*w[k+2];
            d=v0*w[k+3];
        } else {
            a=v0*w[k+4];
            b=v0*w[k+5];
            c=v0*w[k+6];
            d=v0*w[k+7];
        }
        if(v1<0f) {
            a+=v1*w[k+8];
            b+=v1*w[k+9];
            c+=v1*w[k+10];
            d+=v1*w[k+11];
        } else {
            a+=v1*w[k+12];
            b+=v1*w[k+13];
            c+=v1*w[k+14];
            d+=v1*w[k+15];
        }
        if(v2<0f) {
            a+=v2*w[k+16];
            b+=v2*w[k+17];
            c+=v2*w[k+18];
            d+=v2*w[k+19];
        } else {
            a+=v2*w[k+20];
            b+=v2*w[k+21];
            c+=v2*w[k+22];
            d+=v2*w[k+23];
        }
        if(v3<0f) {
            a+=v3*w[k+24];
            b+=v3*w[k+25];
            c+=v3*w[k+26];
            d+=v3*w[k+27];
```

```
        } else {
            a+=v3*w[k+28];
            b+=v3*w[k+29];
            c+=v3*w[k+30];
            d+=v3*w[k+31];
        }
        x[i]=a;
        x[i+1]=b;
        x[i+2]=c;
        x[i+3]=d;
    }
}
public void layer4Back(float[] x,float[] w,float[] e) {
    for(int i=0; i<vecLen; i+=4) {
        int k=8*i;
        float v0=x[i],v1=x[i+1],v2=x[i+2],v3=x[i+3];
        float e0=e[i],e1=e[i+1],e2=e[i+2],e3=e[i+3];
        if(v0<0f) {
            w[k]+=v0*e0;
            w[k+1]+=v0*e1;
            w[k+2]+=v0*e2;
            w[k+3]+=v0*e3;
            e[i]=w[k]*e0+w[k+1]*e1+w[k+2]*e2+w[k+3]*e3;
        } else {
            w[k+4]+=v0*e0;
            w[k+5]+=v0*e1;
            w[k+6]+=v0*e2;
            w[k+7]+=v0*e3;
            e[i]=w[k+4]*e0+w[k+5]*e1+w[k+6]*e2+w[k+7]*e3;
        }
        if(v1<0f) {
            w[k+8]+=v1*e0;
            w[k+9]+=v1*e1;
            w[k+10]+=v1*e2;
            w[k+11]+=v1*e3;
            e[i+1]=w[k+8]*e0+w[k+9]*e1+w[k+10]*e2+w[k+11]*e3;
        } else {
            w[k+12]+=v1*e0;
            w[k+13]+=v1*e1;
            w[k+14]+=v1*e2;
            w[k+15]+=v1*e3;
            e[i+1]=w[k+12]*e0+w[k+13]*e1+w[k+14]*e2+w[k+15]*e3;
        }
        if(v2<0f) {
            w[k+16]+=v2*e0;
            w[k+17]+=v2*e1;
            w[k+18]+=v2*e2;
            w[k+19]+=v2*e3;
            e[i+2]=w[k+16]*e0+w[k+17]*e1+w[k+18]*e2+w[k+19]*e3;
        } else {
            w[k+20]+=v2*e0;
            w[k+21]+=v2*e1;
            w[k+22]+=v2*e2;
            w[k+23]+=v2*e3;
            e[i+2]=w[k+20]*e0+w[k+21]*e1+w[k+22]*e2+w[k+23]*e3;
        }
        if(v3<0f) {
            w[k+24]+=v3*e0;
            w[k+25]+=v3*e1;
            w[k+26]+=v3*e2;
            w[k+27]+=v3*e3;
```

```
                e[i+3]=w[k+24]*e0+w[k+25]*e1+w[k+26]*e2+w[k+27]*e3;
            } else {
                w[k+28]+=v3*e0;
                w[k+29]+=v3*e1;
                w[k+30]+=v3*e2;
                w[k+31]+=v3*e3;
                e[i+3]=w[k+28]*e0+w[k+29]*e1+w[k+30]*e2+w[k+31]*e3;
            }
        }
    }
    public float getRate() {
        return rate;
    }
    public void setRate(float r) {
        rate=r;
    }

    public static void main(String[] args) {
        /*  vecLen, length of training and recall vectors
         *  must be a power of 2 like 16,32,64...
         *
         *  depth, depth of the neural network like 1,2,3...
         *
         *  qIn,qOut for initial and final quasi-random projections.
         *  odd numbers like 11,17
         *
         *  rate, training rate for the neural network
         */
        SWNet4 net=new SWNet4(256,3,11,15,0.01f);
        float[][] cos=new float[256][256];
        float[][] sin=new float[256][256];
        float[] work=new float[256];
        for(int i=0; i<256; i++) {
            for(int j=0; j<256; j++) {
                double t=j*(0.03+0.0002*i);
                cos[i][j]=(float)Math.cos(t);
                sin[i][j]=(float)Math.sin(t);
            }
        }
        for(int i=0; i<10000; i++) {
            float e=0f;
            for(int k=0; k<256; k++) {
                net.train(cos[k],sin[k]);
            }
            for(int k=0; k<256; k++) {
                net.recall(work,sin[k]);
                for(int j=0; j<256; j++) {
                    float diff=work[j]-cos[k][j];
                    e+=diff*diff;
                }
            }
            System.out.println("Error: "+e);
        }
    }
}


public class VecOps {

    final static float MIN_SQ = 1e-20f;
```

```
public static void multiply(float[] rVec, float[] x, float[] y) {
   for (int i = 0; i < rVec.length; i++) {
      rVec[i] = x[i] * y[i];
   }
}

public static void multiplyAddTo(float[] rVec, float[] x, float[] y) {
   for (int i = 0; i < rVec.length; i++) {
      rVec[i] += x[i] * y[i];
   }
}

// x-y
public static void subtract(float[] rVec, float[] x, float[] y) {
   for (int i = 0; i < rVec.length; i++) {
      rVec[i] = x[i] - y[i];
   }
}

public static void add(float[] rVec, float[] x, float[] y) {
   for (int i = 0; i < rVec.length; i++) {
      rVec[i] = x[i] + y[i];
   }
}

public static void add(float[] rVec, float[] x,float y) {
   for (int i = 0; i < rVec.length; i++) {
      rVec[i] = x[i] + y;
   }
}

public static void scale(float[] rVec, float[] x, float s) {
   for (int i = 0; i < rVec.length; i++) {
      rVec[i] = x[i] * s;
   }
}

public static void subtractScale(float[] rVec, float[] x,float[] y, float s) {
   for (int i = 0; i < rVec.length; i++) {
      rVec[i] = (x[i]-y[i]) * s;
   }
}

public static float maxAbs(float[] x) {
   float max=0f,c;
   for(int i=0; i<x.length; i++) {
      if((c=Math.abs(x[i]))>max) max=c;
   }
   return max;
}

// reduce the magnitude by t, if the magnitude is reduced below 0 it is made 0.
// with t=1, 1.5 becomes 0.5, -2.5 becomes -1.5, .9 becomes 0 etc.
public static void truncate(float[] rVec, float[] x, float t) {
   for (int i = 0; i < rVec.length; i++) {
      int f = Float.floatToRawIntBits(x[i]);
      int s = f & 0x80000000;  // get sign bit
      float m = Float.intBitsToFloat(f & 0x7fffffff) - t; //abs(x[i])-t
      if (m < 0f) {
         m = 0f;
```

```
        }
        rVec[i] = Float.intBitsToFloat(Float.floatToRawIntBits(m) | s); // put sign back in
    }
}

public static float sumSq(float[] vec) {
    float sum = 0f;
    for (int i = 0; i < vec.length; i++) {
        sum += vec[i] * vec[i];
    }
    return sum;
}

// Assuming each element of is from a Gaussian distribution of zero mean
// adjust the variance of each element to 1.
public static void adjust(float[] rVec, float[] x) {
    float adj = 1f / (float) Math.sqrt((sumSq(x) / x.length) + MIN_SQ);
    scale(rVec, x, adj);
}

// Approximate square root retaining sign.  Squashing function for neural networks.
// x>=0 fn=sqrt(x);  x<0 -sqrt(-x)
public static void signedSqRt(float[] rVec, float[] x) {
    for (int i = 0; i < x.length; i++) {
        int f = Float.floatToRawIntBits(x[i]);
        int sri = (((f & 0x7fffffff) + (127 << 23)) >>> 1) | (f & 0x80000000);
        rVec[i] = Float.intBitsToFloat(sri);
    }
}

public static void signedSq(float[] rVec, float[] x) {
    int one= Float.floatToRawIntBits(1.0f);
    for (int i = 0; i < x.length; i++) {
        float y=x[i];
        int f = Float.floatToRawIntBits(y) &0x80000000;
        f|=one;
        rVec[i] = y*y*Float.intBitsToFloat(f);
    }
}

// Keep only negatives version of reLU neural network activation function
public static void reLU(float[] rVec, float[] x) {
    for (int i = 0; i < x.length; i++) {
        rVec[i]=(x[i]<0f)? x[i]:0f;
    }
    /*
      for (int i = 0; i < x.length; i++) {
        int f = Float.floatToRawIntBits(x[i]);
        f = f & (f >> 31);  // spread the flag bit over all bits and AND.
        rVec[i] = Float.intBitsToFloat(f);
     } */
}

public static void reLUAndSign(float[] rVec, float[] x,float[] sign) {
    for (int i = 0; i < x.length; i++) {
        int s = Float.floatToRawIntBits(sign[i]);
        int f = Float.floatToRawIntBits(x[i]);
        f = f & ((f&s) >> 31);  // spread the flag bit over all bits and AND.
        rVec[i] = Float.intBitsToFloat(f);
    }
}
```

```
    public static void main(String[] args) {
        float[] x = new float[65536];
        for (int k = 0; k < 20; k++) {
            long t = System.nanoTime();
            for (int i = 0; i < 1000; i++) {
                VecOps.reLU(x,x);
            }
            t = System.nanoTime() - t;
            System.out.println("Per second: " + 1000000000000d / t);
        }
    }
}

public final class Rand {
    final RandInf rng;
    final float sc=-0.999999f/0x80000000;
    public Rand(RandInf rng) {
        this.rng=rng;
    }
    public Rand() {
        this(new RNG());
    }
    public int nextInt() {
        return rng.nextInt();
    }
    public int nextInclusive(int max) {
        long r=0xffffffffL;
        r&=rng.nextInt();
        r*=(long)max+1;
        return (int)(r>>>32);
    }

    public int nextInclusive(int min,int max) {
        long r=rng.nextInt()&0xffffffffL;
        r*=((long)max-(long)min)+1;
        return (int)(r>>>32)+min;
    }

    public int nextExclusive(int bound) {
        long r=rng.nextInt()&0xffffffffL;
        r*=(long)bound;
        return (int)(r>>>32);
    }

    public int nextExclusive(int min,int bound) {
        long r=rng.nextInt()&0xffffffffL;
        r*=(long)bound-(long)min;
        return (int)(r>>>32)+min;
    }

    public float nextFloat() {
        return sc*(rng.nextInt()&0x7fffffff);
    }

    public float nextFloatSym() {
        return sc*rng.nextInt();
    }

    public float nextMutation() {
        int r=rng.nextInt();
```

```
        r&=0xbfffffff;
        r|=0x30000000;
        return Float.intBitsToFloat(r);
    }

    public static void main(String[] args){
      Rand r=new Rand();
      System.out.println("nextInclusive");
      for(int i=0;i<50;i++)
        System.out.println(r.nextInclusive(5));

       System.out.println("nextInclusive2");
      for(int i=0;i<50;i++)
        System.out.println(r.nextInclusive(-5,5));

      System.out.println("nextExclusive");
      for(int i=0;i<50;i++)
        System.out.println(r.nextExclusive(5));

      System.out.println("nextExclusive2");
      for(int i=0;i<50;i++)
        System.out.println(r.nextExclusive(-4,5));

      System.out.println("nextFloat");
      for(int i=0;i<50;i++)
        System.out.println(r.nextFloat());

       System.out.println("nextFloatSym");
      for(int i=0;i<50;i++)
        System.out.println(r.nextFloatSym());

      System.out.println("nextMutation");
      for(int i=0;i<50;i++)
        System.out.println(r.nextMutation());
    }

}

interface RandInf {
    int nextInt();
}

public final class RNG implements RandInf {
    private long s0, s1, s2, s3;
    private static final long PHI = 0x9E3779B97F4A7C15L;
    private static final long MULT=0x3af78c385L;
    public RNG(long seed) {
        s0=seed*MULT+PHI;
        s1=s0*MULT+PHI;
        s2=s1*MULT+PHI;
        s3=s2*MULT+PHI;
    }
    public RNG() {
        this(System.nanoTime());
    }
    public long nextLong() {
        final long t0 = s0;
        final long result = Long.rotateLeft(t0 + s3, 23) + t0;
        final long t = s1 << 17;
        s2 ^= t0;
        s3 ^= s1;
```

```
        s1 ^= s2;
        s0 ^= s3;
        s2 ^= t;
        s3 = Long.rotateLeft(s3, 45);
        return result;
    }

    public int nextInt() {
        return (int)nextLong();
    }
}

public class WHT {
    public static void wht(float[] x) {
        final int n=x.length;
        if(n<=2048) {
            hstepTo8(x,0,n);
            if(n==8) return;
            hstep16(x,0,n);
            if(n==16) return;
            hstepN(x,0,n,16);
            if(n==32) return;
            hstepN(x,0,n,32);
            if(n==64) return;
            hstepN(x,0,n,64);
            if(n==128) return;
            hstepN(x,0,n,128);
            if(n==256) return;
            hstepN(x,0,n,256);
            if(n==512) return;
            hstepN(x,0,n,512);
            if(n==1024) return;
            hstepN(x,0,n,1024);
            return;
        } else {
            for(int i=0; i<n; i+=2048) {
                hstepTo8(x,i,i+2048);
                hstep16(x,i,i+2048);
                hstepN(x,i,i+2048,16);
                hstepN(x,i,i+2048,32);
                hstepN(x,i,i+2048,64);
                hstepN(x,i,i+2048,128);
                hstepN(x,i,i+2048,256);
                hstepN(x,i,i+2048,512);
                hstepN(x,i,i+2048,1024);
            }
            hstepN(x,0,n,2048);
            if(n==4096) return;
            hstepN(x,0,n,4096);
            if(n==8192) return;
            hstepN(x,0,n,8192);
            if(n==16384) return;
            hstepN(x,0,n,16384);
            if(n==32768) return;
            hstepN(x,0,n,32768);
            if(n==65536) return;
            hstepN(x,0,n,65536);
            if(n==131072) return;
            hstepN(x,0,n,131072);
            if(n==262144) return;
            hstepN(x,0,n,262144);
```

```
        if(n==524288) return;
        hstepN(x,0,n,524288);
        if(n==1048576) return;
        for(int hs=1048576; hs<n; hs+=hs) {
            hstepN(x,0,n,hs);
        }
    }
}


public static void whtN(float[] x) {
    final int n=x.length;
    final float scale=1f/(float)Math.sqrt(n);
    if(n<=2048) {
        hstepTo8(x,0,n,scale);
        if(n==8) return;
        hstep16(x,0,n);
        if(n==16) return;
        hstepN(x,0,n,16);
        if(n==32) return;
        hstepN(x,0,n,32);
        if(n==64) return;
        hstepN(x,0,n,64);
        if(n==128) return;
        hstepN(x,0,n,128);
        if(n==256) return;
        hstepN(x,0,n,256);
        if(n==512) return;
        hstepN(x,0,n,512);
        if(n==1024) return;
        hstepN(x,0,n,1024);
        return;
    } else {
        for(int i=0; i<n; i+=2048) {
            hstepTo8(x,i,i+2048,scale);
            hstep16(x,i,i+2048);
            hstepN(x,i,i+2048,16);
            hstepN(x,i,i+2048,32);
            hstepN(x,i,i+2048,64);
            hstepN(x,i,i+2048,128);
            hstepN(x,i,i+2048,256);
            hstepN(x,i,i+2048,512);
            hstepN(x,i,i+2048,1024);
        }
        hstepN(x,0,n,2048);
        if(n==4096) return;
        hstepN(x,0,n,4096);
        if(n==8192) return;
        hstepN(x,0,n,8192);
        if(n==16384) return;
        hstepN(x,0,n,16384);
        if(n==32768) return;
        hstepN(x,0,n,32768);
        if(n==65536) return;
        hstepN(x,0,n,65536);
        if(n==131072) return;
        hstepN(x,0,n,131072);
        if(n==262144) return;
        hstepN(x,0,n,262144);
        if(n==524288) return;
        hstepN(x,0,n,524288);
```

```
        if(n==1048576) return;
        for(int hs=1048576; hs<n; hs+=hs) {
            hstepN(x,0,n,hs);
        }
    }
}

static    void hstepTo8(final float[]x,int start,int stop) {
    for(; start<stop; start+=8) {
        float a=x[start];
        float b=x[start+1];
        float c=x[start+2];
        float d=x[start+3];
        float h0=a+b+c+d;
        float h1=a-b+c-d;
        float h2=a+b-c-d;
        float h3=a-b-c+d;
        a=x[start+4];
        b=x[start+5];
        c=x[start+6];
        d=x[start+7];
        float g0=a+b+c+d;
        float g1=a-b+c-d;
        float g2=a+b-c-d;
        float g3=a-b-c+d;
        x[start]=h0+g0;
        x[start+1]=h1+g1;
        x[start+2]=h2+g2;
        x[start+3]=h3+g3;
        x[start+4]=h0-g0;
        x[start+5]=h1-g1;
        x[start+6]=h2-g2;
        x[start+7]=h3-g3;
    }
}

static    void hstepTo8(final float[]x,int start,int stop,float scale) {
    for(; start<stop; start+=8) {
        float a=x[start];
        float b=x[start+1];
        float c=x[start+2];
        float d=x[start+3];
        float h0=a+b+c+d;
        float h1=a-b+c-d;
        float h2=a+b-c-d;
        float h3=a-b-c+d;
        a=x[start+4];
        b=x[start+5];
        c=x[start+6];
        d=x[start+7];
        float g0=a+b+c+d;
        float g1=a-b+c-d;
        float g2=a+b-c-d;
        float g3=a-b-c+d;
        x[start]=(h0+g0)*scale;
        x[start+1]=(h1+g1)*scale;
        x[start+2]=(h2+g2)*scale;
        x[start+3]=(h3+g3)*scale;
        x[start+4]=(h0-g0)*scale;
        x[start+5]=(h1-g1)*scale;
        x[start+6]=(h2-g2)*scale;
```

```
        x[start+7]=(h3-g3)*scale;
    }
}

static    void hstep16(final float[] x,int start,int stop) {
    for(; start<stop; start+=16) {
        float a=x[start];
        float b=x[start+1];
        float c=x[start+2];
        float d=x[start+3];
        float e=x[start+4];
        float f=x[start+5];
        float g=x[start+6];
        float h=x[start+7];
        x[start]+=x[start+8];
        x[start+1]+=x[start+9];
        x[start+2]+=x[start+10];
        x[start+3]+=x[start+11];
        x[start+4]+=x[start+12];
        x[start+5]+=x[start+13];
        x[start+6]+=x[start+14];
        x[start+7]+=x[start+15];
        x[start+8]=a-x[start+8];
        x[start+9]=b-x[start+9];
        x[start+10]=c-x[start+10];
        x[start+11]=d-x[start+11];
        x[start+12]=e-x[start+12];
        x[start+13]=f-x[start+13];
        x[start+14]=g-x[start+14];
        x[start+15]=h-x[start+15];
    }
}

static    void hstepN(final float[] x,int start,int stop,int hs) {
    while (start < stop) {
        int j = start + hs;
        while (start < j) {
            float a = x[start];
            float b = x[start + hs];
            x[start] = a + b;
            x[start + hs] = a - b;
            start += 1;
        }
        start += hs;
    }
}
/*
    public static void whttest(float[] x){
            final int n=x.length;
            int hs = 1;
   while (hs < n) {
        int i = 0;
        while (i < n) {
          int j = i + hs;
          while (i < j) {
            float a = x[i];
            float b = x[i + hs];
            x[i] = a + b;
            x[i + hs] = a - b;
            i += 1;
          }
```

```
            i += hs;
          }
          hs += hs;
        }
      }

      public static void main(String[] args){
                              float[] x=new float[65536];
                              float[] y=new float[65536];

                              for (int k=0; k<20; k++) {
           long t=System.nanoTime();
           for (int i=0; i<1000; i++) {
             whtN(x);
           }
           t=System.nanoTime()-t;
           System.out.println("Per second: "+1000000000000d/t);
          }
          for(int k=8;k<10000000;k+=k){
                                x=new float[k];
                                y=new float[k];
          for(int i=0;i<x.length;i++){
                                x[i]=(float)Math.random()-0.5f;
                                y[i]=x[i];
                              }
                              System.out.println("K: "+k);
                              whttest(x);
                              whtN(y);
                              for(int i=0;i<Math.min(k,20);i++){
                                        System.out.println("Values true, other:"+x[i]/(float)Math.sqrt(k)+" , "+y[i]);
                              }
                }
                }
                */
}
```